INF-4101C
Rapport

Optimisation des temps de calcul et processeur RISC
Projet “Vision”

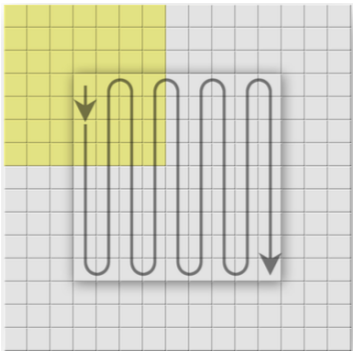
Sinan San Vassia Bonandrini

Table des matières

1. Introduction au „Filtre Médian“

L’objectif principal du filtre médian est de réduire le bruit de l’image en analysant et en traitant les pixels de l'image. En outre, lorsque le filtre médian est utilisé, les contours de l’image sont conservés, donc la détection des contours est plus effective qu'avec les filtres de lissage linéaire ordinaires (par ex. moyenne mobile) [1]. L’idée principale est simple, on regarde une matrice de 3x3 pixels de l'image et on détermine la médiane des valeurs. Ensuite, les valeurs de la matrice sont remplacées par la médiane déterminée. De cette façon, les valeurs particulièrement basses ou élevées sont éliminées (réduire le bruit). Ce procès est répété étape par étape sur l'ensemble de l'image.

Le standard en ‘image processing‘ était pour longtemps Adobe Photoshop, avec un temps d'exécution d'environ O(r) par pixel [1]. Cet algorithme est basé sur l’algorithme de Huang. Il calcule un kernel de base et le fait « bouger » par la suite en additionnant la rangée de pixel devant lui et en retirant celle en arrière puis calcule la médiane des valeurs de ce ce kernel [2]. En utilisant cette méthode, on supprime les calculs redondants et réduit donc la complexité. L'image ci-dessous montre comment le filtre se déplace à travers l'image dans l’algorithme de Huang :



Algorithme de Huang O(r)

2. Rappel des objectifs

L’objectif de ce projet est d’expérimenter des techniques d’optimisation dans le contexte d’une chaîne de traitement d’image embarquée sur la carte SABRE/freescale IMX6 (ARM_Cortex-A9, quad-core) :

1. Optimisation algorithmique (algorithme de complexité inférieure au sens mathématique, structures de données adaptées)
2. Optimisation pour les processeurs RISC (déroulage de boucle)
3. Optimisation pour une utilisation efficace des ressources matérielles : accès aux données, gestion de la mémoire cache, multi-thread

3. Notes d'implémentation

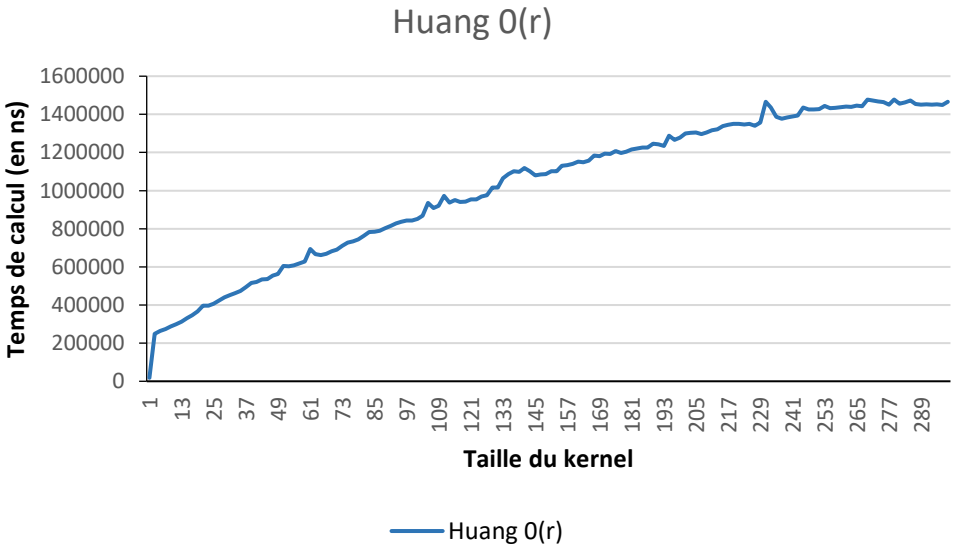
L'algorithme du filtre médian a été implémenté sur une machine virtuelle VMware de distribution Linux ubuntu 64 bits avec 8 Gb de ram et 4 cœurs d'un processeur Intel Core i7 6700K 4.00 GHz de socket 1151 LGA.

Le cache quant à lui possède 3 niveaux :

- L1 Data : 4 x 32KBytes 8-way associative
- L1 inst. : 4 x 32KBytes 8-way associative
- Level 2 : 4 x 256KBytes 4-way associative
- Level 3 : 8 MBytes 16-way associative

4. Évaluation du filtre en version „naïves"

Nous avons commencé avec l'analyse de l'algorithme de Huang. Comme expliqué dans l'introduction, le temps du calcul pur l'algorithme de Huang est O(r), donc on voit qu'en augmentant la taille, le temps de calcul augmente lui aussi.

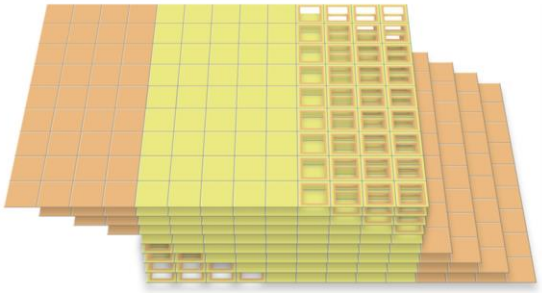


5. Optimisation du filtre médian

Les performances de l'algorithme de Huang sont réduites à la complexité O(r), néanmoins, pour les kernels grands, le temps de traitement est considérablement lent, donc l'application aux filtres avec un radius supérieur à 100 prend trop de temps. Pour améliorer le temps d'exécution par pixel nous avons cherché des solutions et alternatives à l'algorithme de Huang. Nous avons trouvé une idée dans un article de Ben Weiss. L'idée de Weiss pour accélérer le processus est de traiter plusieurs colonnes en même temps donc les calculs redondants entre les matrices deviennent séquentiels et on peut les consolider. En fait, les calculs sont réduits énormément permettant d'augmenter les performances. Plus précisément, l'algorithme de Weiss utilise la propriété distributive des histogrammes, ça veut dire que ce n'est pas nécessaire de traiter chaque histogramme explicitement. On a une

matrice ‘W’, qui est l’union de ‘A’ et ‘B’ (deux disjointes régions de l’image). Il est facile de trouver la médiane de W car on peut utiliser la distributivité des Histogrammes H_A et H_B ($H_W = H_A + H_B$).

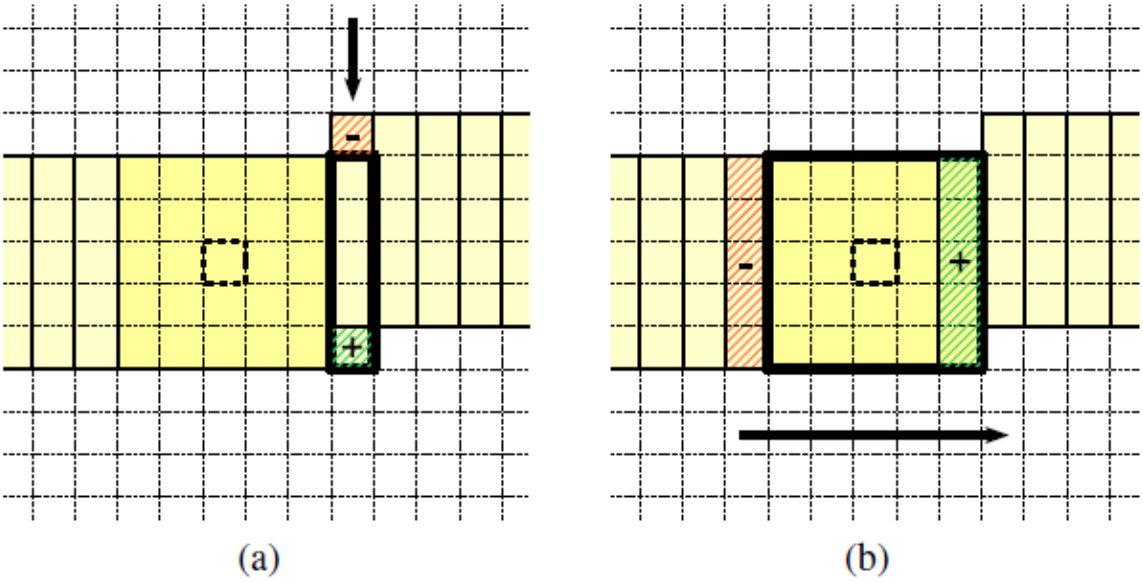
L’approche de Weiss utilise la propriété distributive de former un set d’histogrammes (H*) qui contient des histogrammes partiels. Cela correspond à l’image ci-dessous. Il y a 9 différents histogrammes H_0 á H_8, chaque histogramme comprend l’addition de deux histogrammes partiels. Les trous représentent les pixels qui sont ajoutés au cinquième histogramme (H_4) au milieu et soustrait des autres. La figure montre comment ces histogrammes peuvent être divisés. En effet, il y a 9 colonnes (N = 9) qui sont traitées en même temps.



*Ajouter une ligne des pixels au set des histogrammes H**

5.1. O(1) algorithme reposant sur les histogrammes colonnes (1 par colonne)

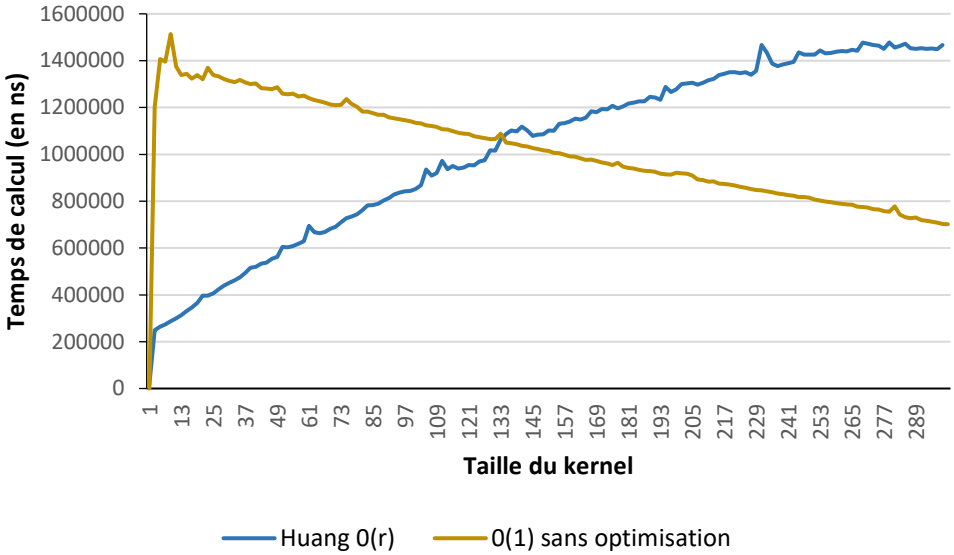
En ce basant sur cette dernière propriété, pour passer de la médiane d’un pixel à l’autre il suffit d’ajouter et de supprimer 1 pixel pour l’histogramme colonne à l’extrémité peu importe la taille du kernel et ensuite ajouter cette colonne à notre histogramme principal [3].



L’algorithme est généralisé à O(1) mais en vérité les passages de ligne sont encore en O(r). Cependant, ces calculs restent négligeables comparés à ceux de la ligne entière.

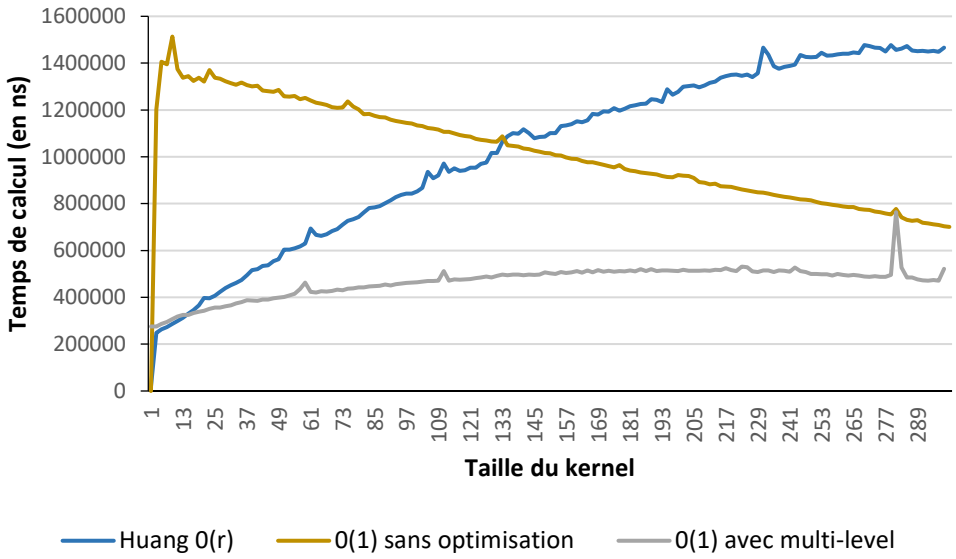
Tous les algorithmes qui vont être vu à partir de maintenant parcourent l'image lignes après lignes optimisant la réutilisation des valeurs comprises dans le cache.

Pour notre première optimisation nous sommes arrivés à améliorer le temps de calcul pour les grandes tailles. On a créé un histogramme pour chaque colonne de l'image donnant pour une taille de kernel supérieur à 133 de meilleures performances que l'algorithme de Huang.



5.2. Multi-level histogrammes

Pour la deuxième optimisation, nous avons implémenté une solution qui utilise des histogrammes à plusieurs niveaux. Pour un histogramme normal, il est composé de 256 valeurs possibles différents pour chaque pixel. Nous avons séparé chacun de ces 256 pixel en groupe de 16 pixels. Il y aura donc maintenant un histogramme représenté par un tableau de 16 valeurs appelé le « coarse » et un tableau 2d de 16x16 appelé le « fine ». Le « coarse » est l'addition de chaque ligne du « fine ».
Ces modifications nous permettent de réduire le nombre d'opération moyen pour calculer la moyenne. Avec en moyenne 128 opérations avant, on obtient maintenant 16 opérations en moyenne (8 pour parcourir le « coarse » et 8 autres pour parcourir le « fine »). De plus, pour la somme des histogrammes, on peut passer le calcul du « fine » lorsque son « coarse » correspondant est égale à 0 économisant du temps. Le graph du temps de calcule montre une énorme amélioration du temps de calcul de l'algorithme.

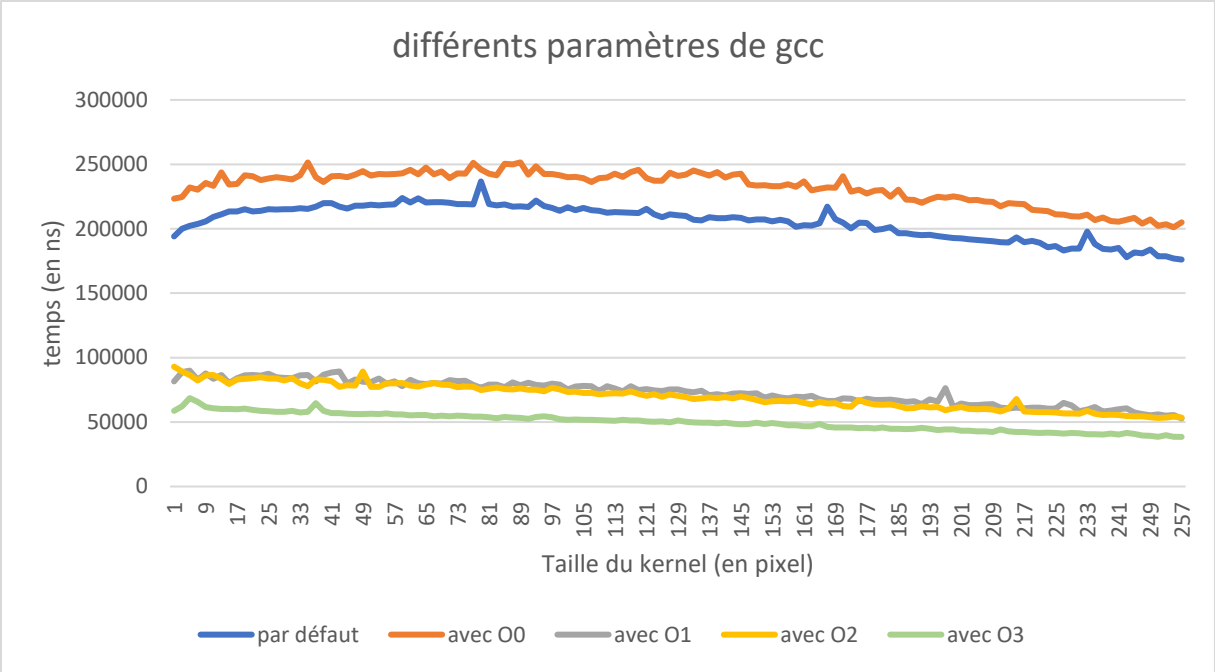


5.3. Parallélisation des instructions

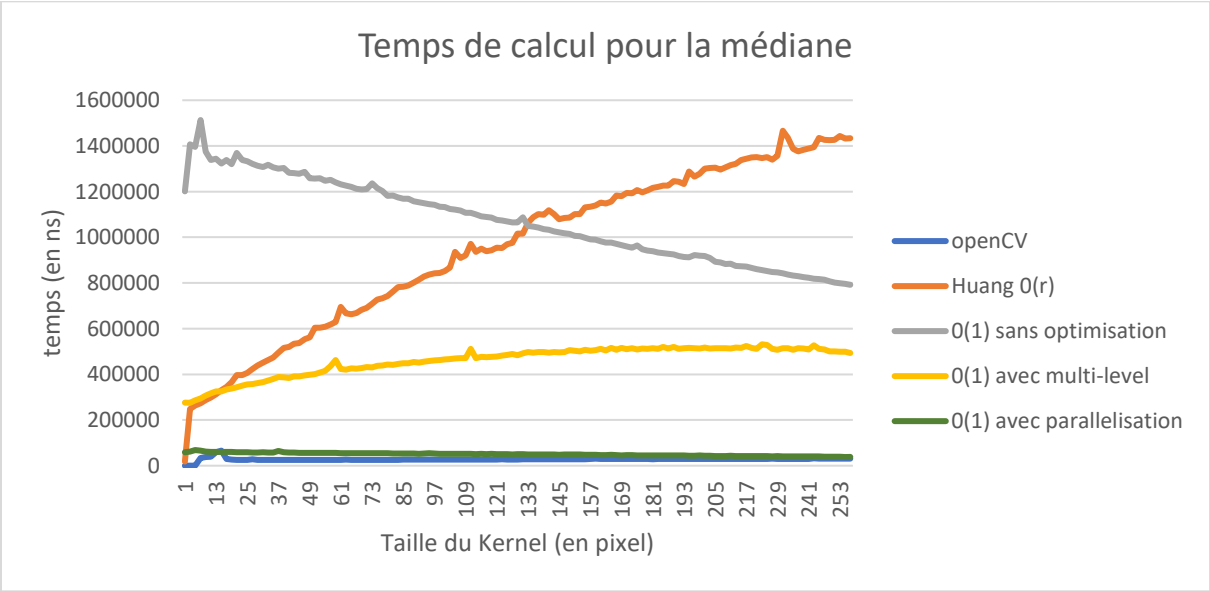
Dans la dernière optimisation nous avons ajouté des instructions SIMD pour la parallélisation de l'addition et de la soustraction des histogrammes. On ajoute les 16 valeurs en même temps pour le coarse et les 16 fines. Donc on a 17 opérations au lieu de 9*16=144 pour avant.
Les instructions SIMD sont des instructions de code assembleur pouvant être utilisé dans des niveaux de langage bas tels que le C et le C++ pour avoir plus de liberté et de maitrise quant au compilateur.

5.4. Optimisation du compilateur gcc

Le compilateur gcc nous offre plusieurs optimisations lors de la compilation. Ces optimisations peuvent apporter une amélioration non négligeable du temps de calcul. Le détail des optimisations de chaque flag est décrit ici : https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Le paramètre s'ajoute avec l'option « -OX » avec X un chiffre lors de la compilation.



6. Résultats obtenus



Nous avons amélioré les performances du filtre médian originel de Huang à chaque étape d'implémentation obtenant un résultat qui est beaucoup plus rapide que ce dernier. A l'aide des optimisations de gcc, nous avons même rattrapé les performances de l'algorithmes de base d'OpenCV. OpenCV est une bibliothèque open-source pour du traitement d'images comme le filtre médian, le filtre bilatéral mais elle contient aussi d'autres fonctions comme l'« Image Tresholding » (convertir les images en images binaires) et le « Template Matching » (la détection d'un objet dans l'image).

Les tableaux suivants montrent la différence de temps de calcul entre notre algorithme final (0(1) avec parallélisation) et les autres algorithmes pour différentes tailles (49,159,257).

Taille du kernel	OpenCV	Huang 0(r)	0(1) sans optimisation	0(1) avec multi-level
49	-55%	+901%	+2142%	+609%

Par exemple pour une taille petite (<49), la différence en temps de calcul entre les algorithmes et notre version final est déjà grande. En effet, l'algorithme naïf (Huang) est 9 fois plus lent que notre algorithme final. Seule la performance d'OpenCV reste 55% plus rapide que notre meilleur filtre.

Taille du kernel	OpenCV	Huang 0(r)	0(1) sans optimisation	0(1) avec multi-level
149	-44%	+2091%	+1948%	+921%

Pour des tailles plus grande la différence augmente. Tandis que le temps de calcul de notre algorithme final reste presque inchangé, celui de l'algorithme naïf augmente fortement jusqu'à obtenir une différence de 2091% puis de 3621% pour une taille de kernel de 149 et 257. De plus, l'écart avec OpenCV est de plus en plus réduit.

Taille du kernel	OpenCV	Huang 0(r)	0(1) sans optimisation	0(1) avec multi-level
257	-21%	+3621%	+1956%	+1179%

7. Pistes d'amélioration

Une nouvelle version aurait dû voir le jour mais par manque de temps, elle n'a pas eu le temps d'être bien débugué et n'est pas prête à être utilisé.
Cette nouvelle version implémentait plusieurs nouvelles optimisations :

- Ajout d'« assert » pour une correction plus rapide des bugs dû à la lecture ou l'écriture de pixels en dehors de l'image.
- Réarrangement du code pour créer des variables en dehors des boucles et les écrire avec le résultat d'un calcul qui serait fait une seule fois au lieu de plusieurs fois auparavant (kernel +1 par exemple).
- Création d'une solution multi-threadé qui calculerait la médiane sur une portion de l'image laissant chaque thread s'occuper d'une partie différente de l'image.

Ces solutions, combinées à celles de gcc, aurait sans doute permis à notre algorithme de devenir plus rapide que celui d'OpenCV.

Les « portions » d'image géré par chaque thread auraient été choisi pour ne pas dépasser la taille du cache et donc avoir un nombre peu élevé de miss améliorant encore plus notre algorithme.

Pour finir, il reste 2 optimisations évoquées dans [3] que nous n'avons pas implémenté :

- La parallélisation du calcul des histogrammes colonne

Actuellement, nous rajoutons et enlevons les pixels qu'il faut pour chaque histogramme colonne avant de calculer la médiane en mettant à jour l'histogramme principale ou histogramme du kernel. Cette étape peut être parallélisé pour mettre à jour plusieurs histogrammes colonnes en même temps cependant c'est difficile à implémenter et même OpenCV a décidé de ne pas le faire.

- Le « conditional update »

Seulement le « coarse » de l'histogramme principale est maintenu à jour et le « fine » n'est calculé que sur demande au lieu de chaque itération. Le segment du « fine » n'est calculé que s'il contient la médiane et seulement avec certains histogramme colonne. Ce saut de calcul de certaines colonnes apporte une optimisation sachant que dans une image les pixels varient rarement brutalement, c'est un phénomène beaucoup plus doux et lisse en réalité.

8. Bibliographie

[1] WEISS, Ben. "Fast median and bilateral filtering." ACM SIGGRAPH 2006 Papers, pp. 519-526.
[2] HUANG, T.S. 1981. "Two-Dimensional Signal Processing II : Transforms and Median Filters" Berlin : Springer-Verlag, pp. 209-211.
[3] Perreault and Hébert, IEEE magazine, « Median Filtering in Constant Time », 2007